ISSN 0316-6295

W. David Elliott and David T. Barnard

Technical Report CSRG-82

July 1977

COMPUTER SYSTEMS RESEARCH GROUP
UNIVERSITY OF TORONTO

Notes on Euclid

edited by

W. David Elliott and David T. Barnard

Technical Report CSRG-82

July 1977

Authors:

Edward G. Aseltine
David T. Barnard
Ernest Chang
Jim des Rivieres
W. David Elliott
Neil E. Kaden
Henry Spencer
David H. Thompson
Ted Venema

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their applications. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the National Research Council of Canada.

Abstract: This report contains papers analyzing various aspects of the programming language Euclid: abstract data types, isolation of machine dependencies, readability and writability, and comparisons of Euclid with Pascal and Modula. This report also includes an index to the Euclid Report.

Acknowledgements: Any contribution "Notes on Euclid" has to make is a tribute to Jim Horning, whose consistent insights and enthusiasm motivated those in the course throughout. Through his lectures our group was privy to the detailed thinking behind Euclid design decisions. We appreciate the efforts of Inge Weber, who patiently edited this report. Ric Holt and Brian Randell provided helpful comments. Professor Niklaus Wirth provided helpful comments and clarified several points about Modula.

Table of Contents

If there's a hole in a' your coats,
I rede you tent it;
A child's among you takin' notes,
And faith he'll prent it.

Burns

Preface: Euclid is a programming language based on Pascal intended for writing verifiable system programs. During the fall of 1976 at the University of Toronto, Jim Horning, one of the Euclid designers, taught CS2124, Topics in Programming Languages. The course comprised a number of weeks of lectures on Euclid and its development by Jim, followed by critiques of various aspects of Euclid presented by students in the course. Those presentations formed the bases of the papers in this report.

The authors of the papers were dealing with a somewhat elusive target. Several different versions of the Euclid Report have appeared. Successive versions took into account criticisms that had been received, including points that were raised by the presentations in CS2124. Accordingly, some of the papers presented here had to be extensively modified, partially by removing significant critical comments, to reflect the current state of the Euclid design. Since the final version of the Euclid Report appeared after the course was completed, the modifications were effected by the editors. We have tried to keep the papers as coherent as possible in spite of this sometimes radical surgery. No approval by any of the Euclid designers, implied or expressed, should be attached to these papers.

At least passing familiarity with both the Pascal and Euclid Reports is assumed throughout most of these papers. [Lampson et al.77] is considered the authoritive version of the Euclid Report (generally referred to herein as "the Euclid Report" or "the Report"), and is pervasive throughout these papers. The Pascal Report ([Wirth71]) as well as the Euclid references [Guttag et al.77], [London et al.77], and [Popek et al.77] should similarly be declared pervasive.

References

[Guttag et al.77]
J.V. Guttag, J.J. Horning, and R.L. London; A Proof Rule for Euclid Procedures; USC Information Sciences Institute Technical Report (May 1977).

[Lampson et al.77]
B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek; Report on the Programming Language Euclid; SIGPLAN Notices 12,2 (February 1977) pp. 1-79.

[London et al.77]
R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J. Popek; Proof Rules for the Programming Language Euclid; Purple Peril Publishing (1977), and to appear in Acta Informatica.

[Popek et al.77]
G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell, and R.L. London; Notes on the Design of Euclid; SIGPLAN Notices 12,3 (March 1977) pp. 11-18.

[Wirth71]
N. Wirth; The Programming Language Pascal; Acta Informatica 1(1971) pp. 35-63.

ABSTRACT DATA TYPES IN EUCLID

by Ernest Chang, Neil E. Kaden, and W. David Elliott

Abstract: Euclid provides features that support abstract data types, but does not strictly speaking provide a true data abstraction mechanism. This paper assesses the data abstraction facilities that Euclid does provide, examines the two ways of instantiating Euclid modules, and discusses other features of modules that the designers of Euclid chose not to include. In particular, the paper addresses the issues of (1) enforceable separation between abstract definition and representation, (2) specifying the relationship between abstract definition and representation, (3) type parameters in modules, (4) operator extensions, and (5) scope restrictions on identifiers.

Contents

1 Introduction

Euclid was designed to support the writing of verifiable system software, with the subgoal, as a near-term project, to differ from Pascal as little as possible. Hence Euclid was expressly not intended to be a research vehicle for new ideas in programming languages. One notable exception the Euclid designers made was in the area of data abstraction.

Virtually all data abstraction mechanisms claim lineage from the Simula 67 class [Dahl et al.68]. Descendent data abstraction mechanisms have not in general been used long enough for us to have much experience with their use, and as a result much of the area of data abstraction still belongs to the realm of research.

Although it was not absolutely necessary for Euclid to provide a data abstraction facility, its designers felt that such a programming tool would greatly contribute to the ability to decompose large programs so that they could be verified with existing verification methods. A data abstraction mechanism also encourages writing modular and structured programs, and thus helps meet other Euclid design goals. ([Horning76] provides an excellent summary of the advantages provided by abstraction mechanisms, both data and procedural.)

2 Euclid Modules

The module in Euclid is an encapsulation mechanism whereby the representation of an abstract object and the implementation of associated operations can be hidden from the enclosing scope. Multiple instances of an abstraction can be realized from the definition of a module type by declaring variables of that type. Closed scopes, and modules in particular, provide explicit control over the visibility of identifiers. Objects, operations, and types defined within the module must be explicitly exported in order to be used; similarly, values of variables declared outside a module must be imported explicitly to be known inside.

2.1 Modules as Abstraction Mechanisms

A data type is defined by a set of values and a set of operations on those values. An abstract data type is a data type with a representation-independent definition. Thus, abstract data types permit access by outside routines only to the abstract values and operations, and not to any of the underlying representation. In this sense, clusters in CLU [Iiskov et al.77] and forms in Alphard [Wulf et al.76] are abstract data types, whereas classes in Simula 67 [Dahl et al.68] are not, since all data structures in the outermost scope of a class are accessible.

For reasons similar to those for Simula 67 classes, Euclid modules are not true abstract data types. Access to identifiers within a module is severely restricted by the import and export clauses as well as the Euclid scope rules, but access is allowed.

Euclid modules can be used, however, to implement abstract data types. This would require additional programmer discipline, unenforceable by the language itself, to ensure that the only entities accessible to outside routines are those abstract entities being defined.

2.2 Two Ways of Instantiating Modules

In Euclid there are two distinct methods of defining equivalent objects and their operations. A module type can be defined, and instances of the module type declared in the enclosing scope. Alternatively, a type definition can be

exported from a module, and objects of that type declared in the enclosing scope.

Since no self-respecting abstract data type paper would be complete without an example of a stack, we will use the stack example to illustrate the two instantiation methods. Consider the following implementation of a bounded stack of integers in which multiple instances of the module will be instantiated. The procedure "Pop" pops the top value from the stack and assigns it to the parameter pased.

```
type Stack(StackSize: unsignedInt) = module
     exports(Pop,Push)
     var IntStack: array 1..StackSize of signedInt
     var StackPtr: 0..StackSize := 0

     procedure Push(X: signedInt) =
          imports(var IntStack, var StackPtr, StackSize)
          begin
          procedure Overflow = ... end Overflow
          if StackPtr = StackSize then
               Overflow
          else
               StackPtr := StackPtr+1
               IntStack(StackPtr) := X
          end if
          end Push

     procedure Pop(var X: signedInt) =
          imports(var IntStack, var StackPtr)
          begin
          procedure Underflow = ... end Underflow
          if StackPtr = 0 then
               Underflow
          else
               X := IntStack(StackPtr)
               StackPtr := StackPtr-1
          end if
          end Pop

     end Stack
```

The user would access the stack by code such as:

```
var A,B: Stack(100)
var Element: signedInt
...
B.Push(3);
A.Pop(Element);
```

Note: Because functions cannot have side effects and "Pop" alters the stack as well as returning a value, we cannot use the more natural form "Element := A.Pop".

Alternatively, if the module "Stack" exported a type definition "Stk", we could have the following module:

```
type Stack = module
     exports(Stk, Pop, Push)
     type Stk(StackSize: unsignedInt) = record
          var StackPtr: 0..StackSize := 0
          var Body: array 1..StackSize of signedInt
          end Stk

     procedure Push(var IStk: Stk(parameter),
                    X: signedInt) =
          begin
          procedure Overflow = ... end Overflow
          if IStk.StackPtr = IStk.StackSize then
               Overflow
          else
               IStk.StackPtr := IStk.StackPtr+1
               IStk.Body(IStk.StackPtr) := X
          end if
          end Push

     procedure Pop (var IStk: Stk(parameter),
                    var X: signedInt) =
          begin
          procedure Underflow = ... end Underflow
          if IStk.StackPtr = 0 then
               Underflow
          else
               X := IStk.Body(IStk.StackPtr)
               IStk.StackPtr := IStk.StackPtr-1
          end if
          end Pop

     end Stack
```

Corresponding user code might be:

```
var S1: Stack                    {instantiates the module}
var A: S1.Stk(100)               {instantiates the object}
var B: S1.Stk(299)               {and another object}
var Element: signedInt
...
S1.Push(B,3)
S1.Pop(A,Element)
```

The differences between these two stack implementations are largely stylistic. There are, however, situations in which the second instantiation method is more powerful, as shown in the following example.

We would like to define a data type "CharString" that contains a procedure "Append" that operates on two strings passed as arguments. The first method of instantiation requires that the data representation be declared inside the module, and requires code such as:

```
type CharString = module
     imports (CharString)
     exports (Append)
     var X: array 1..250 of char
     procedure Append (var S: CharString) =
          imports (X) ...
          end Append
     ...
     end CharString
```

Since "Append" must have access to the representation of the character string "X", it must be located inside the module "CharString". The requirement that the module import itself, in order for "Append" to be able to access the other string, however, presents an illegal situation in Euclid.

Using the second instantiation technique, this problem is easily dealt with:

```
type CharStringModule = module
     exports (CharString, Append)
     type CharString = record ... end CharString
     procedure Append (var S1,S2: CharString) =
          ...
          end Append
     ...
     end CharStringModule
```

with user code

```
var S: CharStringModule
var Sa, Sb: S.CharString
...
S.Append(Sa,Sb)
```

The existence of the two instantiation methods adds to the complexity of the language, especially since combinations of the two methods are possible. The Euclid designers, however, felt that there were situations in which each of the two methods provided a more natural solution. Instantiating modules avoids the bother of re-importing variables of an exported type. And, as we saw above, instantiating a type exported from a module provides capabilities not provided by simply instantiating the module.

3 What Euclid Left Undone

Although modules were a notable exception to the Euclid design guideline of no innovation, somewhat conservative design decisions were made concerning modules. This section discusses areas where the Euclid designers could have provided further module capabilities.

3.1 Separation of Abstraction and Representation

In CLU [Liskov et al.77], Mesa [Geschke et al.77], and Alphard [Wulf et al.76], it is possible to write the specification of an abstract data type as an entity completely distinct from its implementation. In this way, it is possible to change representations quite easily and to implement libraries of data abstractions and implementations, both of which capabilities are highly desirable.

Euclid, however, provides no syntactic mechanism to ensure that this separation is preserved. It is possible in Euclid to implement interchangeable modules for the same abstract specifications, but the specification would have to be textually copied between modules in order to do so. Also, there would be no way within the language of ensuring that the textually copied specifications remain the same.

3.2 Axiomatic Specification

In order to be able to prove that a particular representation of an abstract data type does indeed implement the specified data abstraction, a language must provide a formal means of specifying the relationship between the concrete representation and abstract definition. Although Euclid did not originally provide such a mechanism, the ill-defined abstraction function now fulfills that role, in much the same way that the (somewhat misnamed) rep function does in CLU. In Alphard this relationship between concrete and abstract is specified in the representation section of the form. None of these languages, however, provide an adequate means for axiomatic specification [Guttag et al.76].

3.3 Schemes

Types in Euclid are allowed to have formal parameters. Such parameters are typed constants, but need not be manifest. It is possible to defer fixing the value of a parameter by specifying it as any, unknown, or parameter, but it is not possible to pass types as parameters to a parameterized type.

Mitchell and Wegbreit [76] have coined the term "scheme" as a generalization of parameterized types in which type values can be passed as parameters to the definition mechanism. The instantiation of a scheme is a (possibly parameterized) abstract data type. Thus, for example, passing type "integer" as a parameter to a scheme definition could produce a bounded stack of integers data type. This data type can in turn be instantiated to produce a particular abstract data object. Such facilities exist in both CLU and Alphard, as well as in EL1 [Wegbreit74].

3.4 Operator Extensions

Operator extensibility in Euclid is strictly procedural in nature. The generic operators equality and assignment are identical for all modules, though to be used they must be explicitly exported. A new operator defined on a data type can be viewed only as a routine, not as a new infix or prefix operator.

Operators can be redefined in both CLU and Alphard, and even equality and assignment can be redefined in CLU. Alphard also allows operators to be defined as infix or prefix, which contributes to readability.

3.5 On Inheriting Scope

The Euclid designers took much of the advice of Wulf and Shaw [73] to heart in attacking the problems of aliasing and global variables. Unlike the designers of Gypsy [Ambler et al.77], who discarded the Algol notion of nested scopes, the Euclid designers chose to reinforce Algol-like block structuring with further restrictions. In particular, Euclid requires import lists for closed scopes, prohibits redeclaration of variables within a scope, prohibits "sneak access" to variables via procedure calls, and disallows functions with side effects. Euclid allows different types of access (e.g., read, write) to be associated with an exported or imported variable, as does Alphard.

Both Euclid and Alphard require that an identifier be passed through all intervening scopes in order to be known within an inner scope. (Similarly, in Euclid all ancestors of a machine-dependent module must be made machine-dependent.) Since Gypsy does not permit a hierarchy of routine declarations, there are no intervening scopes that need simply pass an identifier along.

4. Summary

Euclid provides features that support abstract data types, but does not strictly speaking provide a true data abstraction mechanism. With modules, the Euclid designers struck a balance between providing abstraction capabilities and ensuring that the capabilities provided could be fairly easily implemented. In particular, Euclid could have provided further capabilities in the areas of enforceable separation between abstract definition and representation, specifying the relationship between abstract definition and representation, type parameters in modules, operator extensions, and scope restrictions on identifiers.

References

[Ambler et al.77]
A.A. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells; Gypsy: A Language for Specification and Implementation of Verifiable Programs; SIGPLAN Notices 12,3 (March 1977) pp. 1-10.

[Dahl et al.68]
O.-J. Dahl, B. Myhrhaug, and K. Nygaard; The Simula 67 Common Base Language; Norwegian Computing Center, Oslo (1968).

[Geschke et al.77]
C.M. Geschke, J.H. Morris, and E.H. Satterthwaite; Early Experience with Mesa; to appear in CACM.

[Guttag et al.76]
J.V. Guttag, E. Horowitz, and D.R. Musser; Abstract Data Types and Software Validation; USC Information Sciences Institute Technical Report (1976).

[Horning76]
J.J. Horning; Some Desirable Properties of Data Abstraction Facilities; Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, Special Issue (March 1976) pp. 60-62.

[Liskov et al.77]
B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert; Abstraction Mechanisms in CLU; to appear in CACM.

[Mitchell and Wegbreit76]
J.G. Mitchell and B. Wegbreit; A Next Step in Data Structuring for Programming Languages; Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, Special Issue (March 1976) pp. 69-70.

[Wegbreit75]
B. Wegbreit; The Treatment of Data Types in EL1; CACM 17,5 (May 1974) pp. 251-264.

[Wulf and Shaw73]
W.A. Wulf and M. Shaw; Global Variable Considered Harmful; SIGPLAN Notices 8,2 (February 1973) pp. 28-34.

[Wulf et al.76]
W.A. Wulf, R.L. London, and M. Shaw; Abstraction and Verification in Alphard; Carnegie-Mellon University (also USC Information Sciences Institute) Technical Report (1976).

ISOLATION OF MACHINE DEPENDENCIES IN EUCLID

Edward G. Aseltine and Henry Spencer

Abstract: The programming language Euclid was designed for programming such software as operating system kernels, I/O routines, and storage allocators. These applications require access to low level machine details. Euclid makes a serious effort to limit machine dependencies and to isolate them in readily identifiable sections of code. This paper surveys the machine dependencies of Euclid and assesses the effectiveness of the isolation mechanisms.

Contents

1 Introduction

If a language is to allow access to the underlying machine, knowledge and control of representation become essential. The approach taken in Euclid involves language-defined representations for the values of certain types given in terms of a hypothetical binary machine with implementation-dependent parameters. The hypothetical binary machine is somewhat loosely defined, but includes virtually all common ("standard") architectures. The behavior of programs at the machine interfaces, i.e. in sensitive contexts, is then defined in terms of these standard representations.

This paper examines Euclid's machine-dependent features under three categories: those isolated within machine-dependent

modules, those relating to the storage structure of the machine, and others. Each feature is examined with regard to its capabilities, its potential for misuse, and the verification problems it poses.

2 Machine-Dependent Modules and their Contents

Euclid's main device for the isolation of machine dependency is the machine-dependent module. Semantically the machine-dependent module is a normal module, except that it is clearly marked machine dependent and that it may make use of certain machine-dependent features not otherwise permitted. Since the machine-dependent module itself is one of these features, any module containing one must in turn be machine-dependent. However, since variables of types exported from machine-dependent modules can only be manipulated by exported routines, these routines serve to encapsulate the machine dependencies, and a module that imports a machine-dependent module need not itself be machine-dependent.

The features available only within a machine-dependent module are machine-dependent records, variables at fixed addresses, machine code routine bodies, uses of "StorageUnit.Address", type converters involving types with implementation-dependent representations, and extended characters of the form $ddd. With one exception discussed later, the internal representations of values are visible only within machine-dependent modules.

2.1 Machine-Dependent Records

A machine-dependent record is a restricted form of record that allows specification of the position and size of each variable field. Its purpose is to allow clean access to hardware defined data structures such as program status words and device registers.

The machine-dependent record does allow the overzealous bit-twiddler to try to "improve" on Euclid's packed option. This could introduce unnecessary machine dependency into what might otherwise be a machine-independent module. The major obstacle to such misuse is that a machine-dependent record must be declared in a machine-dependent module and must be explicitly exported if it is to be known outside that module. In addition, since one must also export all operations on such a machine-dependent record, any machine dependency is reasonably well confined.

The machine-dependent record does present a minor verification problem in that its structure must be checked to ensure that it accurately reflects the hardware-defined data structure. At some stage, this check must be done by hand since the information needed to check a machine-dependent record definition is derived from the hardware specifications.

2.2 Variables at Fixed Addresses

Euclid permits variables within a machine-dependent module to be declared at fixed machine addresses. This allows access to such things as device registers and interrupt vectors, which have locations fixed by the hardware, and also allows linkage to machine code routines.

However, in allowing access to data and code defined external to Euclid, variables at fixed addresses also provide the ability for Euclid routines to interfere with other software running in the same memory, as well as with compiler space allocation. Abuse is limited by the requirements that such variables (1) be declared only within machine-dependent modules and be explicitly exported, (2) be of a type that has a standard representation, and (3) not overlap any other Euclid defined variable.

Variables at fixed locations suffer from the same verification problem as machine-dependent records in that consistency with the hardware must be verified. Also, difficulties are introduced by hardware-caused side effects inherent in references to certain locations. Such side effects, including triggering of device operations and auto-incrementing, occur in many machines.

2.3 Machine Code Routine Bodies

Euclid allows a routine declared within a machine-dependent module to have a body consisting of a sequence of machine instructions rather than Euclid statements. This provides an escape from Euclid in cases where a machine-specific operation must be performed or especially high efficiency is required.

Abuse of this feature is restricted by several factors. First, access to machine code requires an entire routine definition. Second, such routines must be declared in machine-dependent modules and explicitly exported. Finally, the machine code representation given in the Euclid Report (a series of manifest constants bracketed by the keywords code and end) is inconvenient and this discourages its use. (It is in some ways unfortunate that the Report permits extension of the code syntax.) Verification of machine code routine bodies is generally very difficult.

2.4 Data Addresses

Euclid makes addresses of arrays of "StorageUnit" available in storage allocators. Abuse of this feature is limited by the fact that these are numbers (not pointers) and by the restriction to machine-dependent modules. Verification is potentially very difficult, especially if the storage allocator does inherently machine-dependent arithmetic on such numbers.

2.5 Extended Characters

The use of the $ddd convention for specifying characters provides a way of handling unprintable characters at the cost of specifying them in terms of their hardware representation. The restriction of this feature to machine-dependent modules controls any abuse, and verification should not present major problems.

2.6 Specified Representations

In sensitive contexts the Euclid Report requires that a type's standard representation be specified, either by the Report or explicitly by the implementation. Otherwise, the implementation is free to use any representation, provided it is converted to the specified representation in sensitive contexts.

Abuse of the knowledge of representations is limited by the restricted contexts in which the representation is visible. The only problem associated with this will be discussed in Section 4.2.

3 Storage Structure

The hypothetical Euclid machine has a basic unit of storage allocation called a "StorageUnit". "StorageUnit" is a standard "space-filling" type that consists of a fixed-length sequence of bits on which no operations are defined. The number of bits within a "StorageUnit" is implementation defined and given by the pervasive constant "StorageUnit.sizeInBits".

There are several machine dependencies in Euclid related to a machine's basic storage structure and available in supposedly machine-independent contexts: the built-in components "size" and "alignment", the standard component "StorageUnit.sizeInBits", the standard simple type "AddressType", and the "Index" component of collections. "size" and "alignment", which are implicitly declared components of all types, depend on the storage representation of the type. "StorageUnit.sizeInBits" specifies the (machine-dependent) number of bits in a "StorageUnit". "AddressType" is an unsigned subrange of integer, large enough to hold a full machine address, i.e., a value returned by the function "StorageUnit.Address". The function "Index" is a standard component of collection variables, intended for address manipulation in zones, that accepts a pointer to the collection's object type and returns an integer version of the pointer. These features are useful mainly for storage allocation, notably in programmer defined zones.

The possible abuse of these features arises from their being machine-dependent quantities that are available without restriction. A routine that is not marked as machine-dependent may make use of them, thus becoming in reality machine-dependent. A solution, albeit a somewhat stringent one, would be to allow access to these quantities only within machine-dependent modules.

4 Odds and Ends

The remaining machine dependencies in Euclid include the ordering and representation of the character set, type converters, and some more machine defined quantities.

4.1 Character Set

Because there is no universally accepted standard, Euclid deliberately does not specify the character set to be used in its implementations. This means that the ordering of variables of type "char" is implementation dependent, although use of this ordering is not restricted to machine-dependent modules. Thus, use of ordering in (1) comparisons of characters, (2) subranges of characters, or (3) the "Chr" or "char.Ord" functions will in general result in non-portable programs. The programmer can assume only

```
$a < $b < ... < $z
$A < $B < ... < $Z
$1 = char.Succ($0)
...
$9 = char.Succ($8)
```

(Despite the best of intentions about the Euclid Report and the Euclid Proof Rules defining the same language, the ordering of the alphabet and of the decimal digits as characters is specified in the Proof Rules, but not in the Report itself.) As mentioned above, use of extended characters of the form $ddd is restricted to machine-dependent modules.

4.2 Type Converters

Euclid provides for breaches of the type system via type converters, but attempts to restrict their use by making them cumbersome to work with. The source type and target type must have the same size, and neither can be parameterized. If either type has an implementation-dependent representation, the type converter can only appear in a machine-dependent module. A type converter may not be referenced in the scope in which it is automatically declared, and must be explicitly imported into any inner closed scope that references it.

4.3 More Machine-Defined Quantities

Euclid provides standard types "unsignedInt", "signedInt", and "string". The quantities "unsignedInt.last", "signedInt.first", "signedInt.last", and "stringMaxLength" are implementation-dependent (if not machine-dependent), but their use is unrestricted. "unsignedInt" and "signedInt" could be eliminated in favor of explicit subranges. Removal of "stringMaxLength", which is needed as a bound in the definition of "string", however, would be very difficult.

5 Summary

As a system programming language, Euclid must make low level machine features available. For portability, however, these machine-dependent features should be isolated and readily identifiable. Isolation of machine dependencies has not been completely achieved. The machine-dependent module isolates many machine-dependent features, but the language leaves a number of lcopholes that will make transporting and verifying Euclid programs difficult.

READABILITY AND WRITABILITY IN EUCLID

Jim des Rivieres and Henry Spencer

Abstract: Neither readability nor writability of programs were major design goals of Euclid, yet both are obviously important considerations in determining how effectively the language can be used. This paper considers the influence of various Euclid features on readability and writability.

Introduction

Euclid's principal goal of verifiability has a major impact on the desired attributes of readability and writability. To further verification, more of the information needed for understanding and maintenance is made explicit in Euclid programs; this information should also greatly improve readability. On the other hand, the Euclid designers expected to sacrifice some writability to improve verifiability, since more program text is required to make this information explicit.

This paper discusses the influence of individual Euclid features on readability and writability. We do not attempt to define these notions formally, since the boundary is somewhat fuzzy. (How exactly does one classify features which ease the writing of readable code?) We concentrate on those aspects that are comparatively new in Euclid, and avoid discussion of such questions as why the *case* statement is superior to the computed *goto*.

The order of discussion and the numbering of sections correspond to those of the Euclid Report.

3 Notation, Terminology, and Vocabulary

3.1 Vocabulary

Long identifiers containing breaks, such as those allowed by Euclid, can significantly increase readability. Euclid eliminates a possible source of errors in such identifiers by ignoring breaks when checking for redeclaration, thereby

prohibiting the use of identifiers that are equivalent except for breaks.

There are some unresolved points when using the change from lower to upper case to indicate a break. It will probably be a characteristic (i.e., frequently recurrent) error of upper/lower case to forget whether the first letter of an identifier is upper or lower case. While the Report carefully specifies that the vocabulary is not the character set, it would still be useful for language standardization to state in what case keywords should be written.

The comment convention of Euclid is quite orthodox and has the often cited problems of bracketing comment symbols. Bracketing comment symbols have the writability problem of "runaway comments" if the right comment delimiter is omitted. Also, their writability is inferior to that of a symbol-to-end-of-line convention for short comments since they require a closing bracket. For large blocks of text, readability would improve if the comment delimiters were more conspicuous.

3.3 Lexical Structure

Automatic semicolon insertion should make Euclid programs easier to write by eliminating the characteristic error of omitting trailing semicolons.

The end brackets for all control constructs, blocks, records, and modules improve readability by making the program structure more explicit, and as such should be less error-prone. end brackets for control constructs similarly improve writability by eliminating special cases when adding or deleting statements.

4 Identifiers, Numbers, and Strings

The representation for octal and hexadecimal constants is good except for a characteristic writability error with hexadecimal numbers. Because the first digit of a hexadecimal number must be a decimal digit, "E0#16" must actually be written "0E0#16".

The character constant convention is poor in two respects. Readability is impaired by the density of "$", which overpowers most other characters; a low-density character like "'" would have been preferable. There is also a writability problem due to the unusual, unfamiliar convention.

5 Manifest Constants

The availability of expressions as manifest constants improves readability by making the relationship between constants explicit and by reducing the need for extra variables. Expressions as manifest constants also improve writability by

reducing the need for hand calculations, and improve both readability and writability by facilitating parameterization of types.

6 Data Types

Free variables are strictly prohibited from type declarations by the Euclid Report. There are, however, situations where free variables are absolutely necessary, such as collection names in the declaration of pointer types. This is a distinct problem with at least the wording of the Report, and is being fixed.

The built-in facilities for getting at attributes of types, e.g. "unsignedInt.last" and the "IndexType" component of array types, make Euclid programs easier to read by providing a consistent way of referring to attributes for related types. The rule for when to capitalize built-in components, however, seems capricious at best. Built-in facilities also make Euclid programs easier to write by providing facilities that otherwise would have to be constructed by the user, in some cases clumsily.

The "type-of" component of variables and constants is named "itsType" instead of the preferable "Type" because type is a reserved word.

6.1 Simple Types

By restricting the programmer to more straight-forward constructs, the absence of type and routine variables is a great asset for understandability.

6.1.1 Enumerated Types

The uniformity of reference to "T.Succ(X)" and "T.Pred(X)" must be weighed against the clutter introduced by requiring the type names.

6.2.1 Array Types

Multi-dimensional arrays must be formed by having arrays of arrays. Although the decision to eliminate multidimensional arrays is understandable given the types of programs for which Euclid was designed, their absence may have minor negative effects on both readability and writability.

6.2.2 Record Types

Allowing constants in records allows relevant information to be grouped together, thus improving readability.

There are some possible characteristic errors associated with record variants. A minor one is writing else instead of otherwise. Two more serious ones, arising from not really thinking of variants as parameterized types, are forgetting to include the variant tag in the parameter list and forgetting to give a parameter list when declaring variables of type "T(any)".

6.2.3 Module Types

The module serves as an encapsulation mechanism, and as such potentially offers great gains in readability. There are, however, some problems with the two distinct ways of instantiating modules (as discussed in the "Abstract Data Types in Euclid" paper): modules exporting routines and types but having no local variables exported, and modules exporting no types but exporting variable components. The two different usages, and the unfortunate possibility of combinations, hurt readability by increasing the complexity of the language.

The explicit flagging of machine-dependent modules makes machine dependencies more clearly identifiable, and thus increases readability. (A discussion of how easily identifiable the different machine dependent features are appears in the "Isolation of Machine Dependencies in Euclid" paper.)

6.2.4 Machine-dependent Records

The parentheses surrounding the at clause are a good, readable way of introducing a third entity into the Pascal-style declaration.

6.2.5 Set Types

xor and * seem to be unnatural choices for set opeator symbols.

6.2.6 Pointer and Collection Types

The requirement of declaring a collection will mean more writing in simple cases of dynamic variable allocation. Explicit facilities for storage allocation, however, should make storage allocation routines easier to write in general, although user-defined storage allocation routines will still be complex routines to write. The resulting code should be more understandable due to the standard set off primitives with which to construct such routines.

The use of collections improves readability by allowing the user to group related dynamic variables, e.g., collections named "BritishCitizens" and "CanadianCitizens".

The availability of a built-in reference count mechanism improves both readability and writability by obviating explicit deallocation where only simple reference counting is needed.

6.3 Parameterized Types

The use of parameterized types improves writability by eliminating the need for writing repetitive code to handle similar functions for different data types. Parameterized types thus solve the Pascal problem of needing separate procedures to perform the same function for arrays of different ranges [Habermann73]. The strong typing of parameters helps control possible abuse of parameterized types.

6.4 Type Compatibility

The type compatibility rules given in the Report, although precise, are hard to understand. The Report should perhaps include a simpler overview discussion such as that in [Popek et al.77] to explain the purpose of the rules.

7 Declarations and Denotations of Constants and Variables

For arguments similar to those for keyword versus positional parameters, the syntax of structured constants is hard to read and error-prone to write, especially for constants of record types. Also, there is no way to specify that the length of an array constant is to be taken from the length of the value list.

The initialization of variables in declarations makes programs more readable as well as writable by putting relevant information together.

Bindings improve readability by binding identifiers to the relevant data object, thus improving clarity and reducing clutter. When used like the Pascal *with* statement, bindings also improve writability by permitting abbreviation. The lack of uniformity between the single binding syntax and binding-list syntax hurts writability (specifically modifiability).

7.3 Scope Rules and Importing

The complexity of the scope rules will cause problems in writing Euclid programs.

In general, *imports* lists improve readability at the expense of writability. Forcing the user to specify all imported variables makes interfaces explicit, but requires extra code. If "X" imports "Y" and "Y" imports "Z", "X" must also import "Z". This will make the program harder to modify, as well as adding to the length of the *imports* list. Since imported items are constant by default, the clause "*imports* (*var* X,Y)" imports "Y"

as a constant, which is not what programmers used to Pascal declarations would expect.

pervasive seems a mixed blessing. Its use can improve readability by reducing imports lists to the important items. The inability to override a pervasive declaration in an inner scope, however, makes it impossible to write an inner scope independently of its enclosing scope.

8 Expressions

The use of operator precedences as well as the particular levels should reduce high-persistence APL-like precedence errors (see [Gannon and Horning75]), help eliminate parentheses, and generally contribute to reading and writing clarity.

8.1.3 Relational Operators

The ability to test explicitly whether a value is within range with the in operator is a definite asset to both readability and writability.

8.1.4 Other Operators

The logical implication ("->") operator makes complex assertions both more readable and more writable.

The conditional and and or operations can eliminate nested if statements, and thus contribute to both reading and writing clarity. For programmers unfamiliar with them, however, they initially reduce readability since they are not as explicit about control flow as are nested if's.

9 Statements

The inability to determine from a call which arguments are variable parameters, and hence subject to modification, hurts readability.

9.1.3 Escape Statements

Proper use of escape statements should generally make programs both easier to read and easier to write by providing more natural specification of program control. Escapes provide the potential for misuse, however, if the user is too tricky.

The when clause improves both readability and writability by providing a clearer notation for escape conditions.

9.1.4 Assert Statements

Although they may cause run-time checks, assertions are a major aid to readability by concisely describing the meaning of a routine. The ability to make the body of an assertion a comment permits using non-Euclid constructs in assertions, helping both readability and writability by allowing more descriptive assertions.

9.2.1 Compound Statements and Blocks

Allowing declarations in compound statements can increase both readability and writability by putting declarations closer to their use. This also allows more precise use of bindings.

9.2.2 Conditional Statements

The provision of elseif can reduce deep nesting of if's, and as such improves both readability and writability. It does introduce the characteristic error of accidentally putting a blank between the "e" and the "i".

As with variant records, there is a possible characteristic error of writing else rather than otherwise in the case statement.

9.2.3 Repetitive Statements

The ability provided by escape statements to write a loop executed "n and a half" times is an asset for both readability and writability, as mentioned above. Generators similarly provide more natural specification of program control.

10 & 11 Routines

The use of positional, instead of keyword, parameters hurts readability somewhat and could be more error-prone. On the other hand, positional parameters are less verbose. The use of parameter should make calls easier to read.

The elimination of side effects in functions may make some functions less natural to write, but is a big help for understandability.

If a routine has a forward definition, the parameter and import specifications must be placed in the forward definition itself, not where the routine is actually defined. Thus, information about a routine can be greatly separated, hampering readability.

References

[Gannon and Horning75]
J.D. Gannon and J.J. Horning; Language Design for Programming Reliability; IEEE Transactions on Software Engineering SE-1,2 (June 1975) pp. 179-191.

[Habermann73]
A.N. Habermann; Critical Comments on the Programming Language Pascal; Acta Informatica 3 (1973) pp. 47-57.

EUCLID AND PASCAL

Ted Venema and Jim des Rivieres

Abstract: Euclid was intended for writing system programs that could be verifiable by state-of-the-art verification methods. Since verification was not an explicit goal in the design of Pascal, it is not surprising that this gave rise to differences between the two languages. The Euclid designers intended to change Pascal only where it fell short of this goal. This paper examines differences in the two languages in the light of this objective. These differences are roughly grouped under the headings verification, system programming, and user-oriented changes.

Contents

1 Verification

The basic method for verifying Euclid programs is an inductive assertion one similar to that used for Pascal [Hoare and Wirth73]. Those features present in both Euclid and Pascal, but different in Euclid due to the verification requirement, will be the focus of attention in this paper. Constructs added to Euclid specifically for verification will not be discussed, since they in general have no Pascal counterpart.

1.1 Aliasing

Aliasing is the use of two names to refer to the same variable. Aliasing occurs within most block structured languages and is illustrated in the following Pascal example:

```
var I:integer;
procedure P(var J:integer);
     begin
     J:=2*I
     end;
I:=2;
P(I)
```

Within the invocation of "P", both "I" and "J" refer to the same variable.

For the example given above, the verifier might be required to prove that "P" does not affect the value of "I". Examination of the procedure "P" would indicate that this is indeed the case. In particular, an application simply of the axiom of assignment as specified by Hoare [69] shows that for any postcondition "Q", syntactically substituting "2*I" for every free occurrence of "J" in "Q" places no restrictions on "I", thereby allowing the verifier to assume that "I" is unchanged. However, in the example if the formal parameter "J" is bound to the actual parameter "I", then the assertion that "I" remains unchanged during the execution of "P" is false.

1.1.1 Open and Closed Scopes

The elimination of aliasing caused several major changes from Pascal. The first of these is the distinction between open and closed scopes, together with associated scoping rules for each. Pervasive constants are available in all enclosed scopes, open or closed. Open scopes are begin end blocks, component statements of if statements, elements of case statements, and loop or for statements. When an open scope is entered, the identifiers available in the immediately enclosing scope are automatically available in the open scope.

When a closed scope (routine or module) is entered, no variables are automatically available from the enclosing scopes;

variables must be explicitly imported. Conceptually, at the time of a routine call or module instantiation, there is an "extended parameter list" composed of the union of the actual parameters, imported variables and constants, and pervasive constants. A routine call is legal only if the variables in the extended parameter list do not overlap. This requires that no two identifiers of variables in the extended parameter list be the same. Recoding the earlier example in Euclid, we have

```
var I:signedInt
procedure P(var J:signedInt) =
     imports (var I)
     begin
     J:=2*I
     end
I:=2
P(I)
```

The procedure invocation "P(I)" is illegal since the extended parameter list (I,I) contains duplicate elements.

1.1.2 Elimination of Aliasing due to Overlapping Variables

Unfortunately, scoping restrictions are not sufficient to resolve all occurrences of aliasing. Consider, for example, the following Euclid program:

```
procedure Assign(var Left,Right:signedInt) =
     begin
     Left:=Right
     end
```

To avoid aliasing, any call to "Assign" where "Left" and "Right" are bound to the same or overlapping variables must be prohibited. In many cases the compiler can check this. However, if "A" is an array, "Assign(A(I),A(J))" is only legal when "I" $\neq$ "J", which is not generally determinable at compile time. Euclid requires that the compiler generate a legality assertion for the verifier if it is unable to determine that "I" $\neq$ "J" for all executions of the call. This assertion will be compiled into code if the checked option is specified.

The aliasing problem in closed scopes arises from passing parameters by reference rather than by value. If passed by value, the parameter would be constant in the closed scope, and no aliasing problem exists.

A collection is a group of variables of the same type. Just as an index value uniquely determines an element of an array, so a pointer into a collection uniquely determines a variable of the collection. By requiring all dynamic variables to be allocated as part of a collection and by enforcing the collection/array analogy, Euclid severely constrains the use of pointers and hopefully avoids the usual verification pitfalls associated with

them. As with arrays, there remain aliasing problems that cannot be resolved at compile time.

1.2 Pascal Verification Holes

Pascal record variants are not type-safe in that the type of the variant for a particular record instance is not known at compile-time. Pascal allows assignment to a record variant without checking whether that assignment is valid. For example:

```
type Person =
     record
          Name:alfa;
     case Sex:(Male,Female) of
          Male:(Weight:integer);
          Female:(Height:integer);
     end;
var Mary:Person;
Mary.Sex := Female;
Mary.Weight := 180
```

The second assignment is illegal since the current record variant is female. Although in this case a smart compiler could spot the error, this is not generally possible at compile-time. Euclid eliminates this problem by allowing assignment to a variant only within a discriminating case statement. A similar record structure in Euclid would be:

```
type Person(Sex:(Male,Female))=
     var Name:string
     case Sex of
          Male => var Weight:unsignedInt end Male
          Female => var Height:unsignedInt end Female
     end Person
var Mary:Person(Female)
with X bound to Mary case Sex of
     Male => ...
     Female => X.Height:=180
end case
```

The compiler can easily ensure that illegal assignments cannot occur.

Here, as elsewhere in Euclid, a design decision was made to ensure that only local information about the program was needed for verification. Similarly, the compiler need only check each closed scope boundary to ensure that proper control over visibility of identifiers is maintained.

Another aspect of Pascal that presents verification problems is procedure parameters. For example, the following is legal in Pascal:

```
procedure P(procedure Q; Which:integer);
     begin
     if Which = 1 then
          Q(X,Y)
     else
          Q(X)
     end
```

Guaranteeing the correctness of this program involves checking that for all calls to "P", the procedure "Q" has the appropriate number of parameters for the corresponding run-time value of "Which". Euclid avoids this problem by not allowing procedures to be passed as parameters. For similar reasons, types are disallowed as parameters. Both routine and type parameters were felt to be difficult constructs that provided little advantage at a high cost in verification. Some of the uses of routine and type parameters are subsumed by modules, generators, and zones.

1.3 Functions and Side Effects

Pascal recommends against the use of functions that have side effects, but does allow them. Although it is possible to verify programs that allow functions (and hence expressions) to have side effects, this becomes much more difficult. In particular, the axiom of assignment specified by Hoare [Hoare69] must be made much more complicated. As a result, side effects in functions were disallowed in Euclid.

While disallowing side effects in functions makes Euclid programs easier to verify, there are still cases where such side effects seem necessary for a "natural" solution. For example,

```
while Searching do
     begin
     S
     end
```

where "Searching" is a function that returns a Boolean value indicating whether or not elements in the series remain. If the value is true, "Searching" has the side effect of setting a global variable to the value for the next element.

A possible solution that avoids side effects is to factor the function into two parts: a Boolean function "Anymore" that determines if there are any more elements and a procedure "Getnext" that gets the next element. Since in many cases searching involves first attempting to get an entry and then determining if there are any left, this would be coded as:

```
Getnext
while Anymore do
     begin
     S
     Getnext
     end
```

But repeating the call to "Getnext" is undesirable. This can be avoided by using, instead of a while or a repeat until statement, a looping construct where the exit statement can be placed anywhere in the loop. As a result, Euclid has discarded the Pascal while and repeat until statements in favour of a loop with possibly multiple exit statements. This example in Euclid then becomes

```
loop
Getnext
exit when not Anymore
S
end loop
```

With this looping construct, the restriction of not allowing functions with side effects in cases such as this becomes not only tolerable, but preferable since the Euclid version seems to be the easiest to understand of the three versions presented. Euclid also provides generators for more general iteration.

2 System Programming

2.1 Machine-Dependencies

Since Euclid was designed for programming such systems as operating system kernels, it was necessary to provide access to the underlying machine. Pascal does not allow such access. The approach to machine-dependent facilities in Euclid reflects the widely held opinion that machine dependencies should be the exception to the rule. The unnecessary use of machine-dependent facilities in Euclid is discouraged, principally by cumbersome mechanisms, and the necessary uses are in general isolated and readily identifiable. The "Isolation of Machine Dependencies in Euclid" paper provides a more complete discussion of the machine dependencies of Euclid and assesses the effectiveness of the isolation mechanisms.

2.1.1 Standard Representation of Values

In both Pascal and Euclid programs, the internal representation of values is generally irrelevant. The system software for which Euclid was designed, such as operating system kernels, however, requires access to the underlying machine. Euclid defines the representations of values of certain types in terms of a hypothetical binary machine with implementation-

dependent parameters, and then defines the bahavior of programs in sensitive contexts in terms of these standard representations.

The only types given language-defined standard representations are those that have a "natural" implementation on the hypothetical machine. These are all enumerated types (including "Boolean" and "char"), subrange types with a non-negative first value (including "unsignedInt"), sets of enumerated or subrange types, "StorageUnit", and all unpacked array types.

The user can define the internal representation of records by using machine-dependent records. These records can have both constant and variable components. Variable components must be given a starting displacement in "StorageUnit"s and a range of bits, and their type must have a standard representation. Constant components can have neither size nor position specified.

2.1.2 Fixed Address Variables

Euclid allows a variable to be declared at an absolute machine address provided the type of the variable has a standard representation. The address must be a manifest non-negative integer. For example,

```
var Psw (at 177776#8) : ProgramStatusWord
```

declares a variable at memory location 177776 (octal).

2.1.3 Machine Language Code

Access to the machine's instruction set is available in a very simple albeit crude form. Instead of a procedure or function body, a sequence of machine language instructions may be written. The syntax provided is minimal: a list of manifest integer constants can be placed between the keywords code and end. The meaning of these constants is implementation defined. This is the only mechanism by which Euclid programs can access the machine's instructions or invoke a non-Euclid routine. The parameters and results of machine coded routines must have known representations.

2.2 Type Converters

Type converters allow breaches of the type system so that a value of one type can be used as a value of some other type. Both types must have the same size values, and both must have standard or implementation defined representations.

2.3 Dynamic Storage Allocation

Pascal provides dynamic storage allocation, but does not allow the user to define his own allocation scheme. Euclid provides programmer control over storage allocation via zones. The zone has three special components: an internal variable from which the space is obtained, an "Allocate" procedure, and a "Deallccate" procedure.

The zone deals only with blocks of "StorageUnit"s and is not ccncerned with the types of the objects being allocated. The user can define collections that are allocated in a particular zone. The collection components "New" and "Free" provide this interface.

Collections are allowed whose storage deallocation is handled simply by reference counts. Reference counted collections are straight-forward to implement due to the distinction between "New"/"Free" and "Allocate"/"Deallocate".

Although the attempt to provide the user with dynamic storage allocation facilities is commendable, there are several unsatisfactory aspects of the dynamic allocation mechanism in Euclid:

(1) It is impossible to define a zone that can periodically compact its heap since there are pointers into it that are only known outside the zone.

(2) A collection is allocated in a zone, and a zone must have ancther collection as one of its components. Fortunately, if the zone is not specified for a cclIection, a standard system zone is supplied. In most cases, the zone never need invoke "New" or "Free" for its collection. Rather, the collection's "Index" component will be used to allocate pieces of a declared array variable.

(3) Because "New" and "Free" can increase the amount of storage required for a value of the object type (e.g., by adding reference counts), it may be impossible to check for insufficient storage.

2.4 Minor Changes

(1) Input/output facilities are not present, since Euclid is intended as a tool for constructing such system software. However, I/O primitives similar to those of Pascal could be constructed using machine-dependent modules. The differences would be as follows:

    (a) A module "M" could define I/O for only a single, possibly parameterized type.

(b) The Pascal procedures "Get", "Put", etc. would have to be referenced as "M.Get", "M.Put", etc.

(c) The Pascal procedures "Read" and "Write" take a variable number of arguments of various types. In Euclid several simple procedures such as "M.ReadChar", "M.WriteInt", and "M.WriteString" would have to be used.

(2) Only integer arithmetic is supplied in Euclid.

(3) Inline expansion of Euclid routines can be indicated.

(4) Nonprinting characters are allowed in Euclid string constants.

(5) Hexadecimal and octal numbers are allowed in Euclid.

(6) Pascal's abbreviated syntax for multi-dimensional arrays is not provided in Euclid. Although only singly dimensioned arrays are allowed in Euclid, the object type of an array can in turn be an array type, thus providing an arbitrary number of dimensions.

3 User-oriented Changes

3.1 Data Types

The powerful data types and associated type-checking rules of Pascal form the basis of Euclid's data type facilities, although Euclid provides for much stronger type-checking. All of the data structuring methods of Pascal, with the exception of files, are provided in Euclid. What follows is a discussion of some of the enhancements made to Pascal's data type definition mechanisms.

3.1.1 Record and Module Types

Pascal records have been generalized into Euclid records and modules. Euclid's records may contain both constant and variable components. Variable components may have an initial value clause.

Perhaps the most significant extension made to Pascal is the module structure, which can also include type, procedure, and function components. Euclid modules are discussed in some detail in the "Abstract Data Types in Euclid" paper.

3.1.2 Parameterized Types

Perhaps the most frequently cited deficiency of Pascal is caused by the requirements that the index types for formal and

actual array parameters be the same and that all array bounds be compile-time evaluable. This makes it impossible, for example, to have a general sorting or matrix multiplication routine. This situation is rectified in Euclid by allowing the bounds of subrange types to be given by runtime constants. An array with such a subrange for an index type would have its array bounds computed upon entry to the scope.

More generally, types in Euclid are allowed to have formal parameters. Such parameters are typed constants, but need not be manifest constants. Thus, fixed-length vectors of integers could be defined as follows:

```
type IntVector(Length:unsignedInt)
     = array 1..Length of signedInt
```

A 10 element vector could then be declared

```
var Vec:IntVector(10)
```

By using parameterized types, one can write a routine that accepts or returns data objects containing variable-sized arrays, without losing the advantages of strong type checking.

3.1.3 Standard Type Components

Euclid automatically defines several built-in components for each type declared. These components are inherited by variables and constants of the type. For example, all enumerated and subrange types "T" have components "T.first", "T.last", "T.Succ", "T.Pred", and "T.Ord"; all array types have "IndexType" and "ComponentType" components. The parameters of a parameterized type are alsc components. Thus, type dependencies can be indicated without having to introduce extraneous type identifiers.

For example, suppose we have a Pascal array type "A" defined by

```
type A = array[ 1..100] of T;
```

If we wished to declare a variable "X" that has the same structure as a component of array "A", we would have to write

```
var X : T;
```

From this declaration of "X" there is no explicit connection between "X" and the array type "A". Euclid allows this relation tc be stated explicitly, for example

```
type A = array 1..100 of signedInt
var X : A.ComponentType
```

Similarly if we wanted a variable "I" that took the values of the indices of "A" together with one more value at the end, we could write

```
var I : A.IndexType.first .. A.IndexType.last + 1
```

A type's components are automatically inherited by its variables and constants. Variables are also given a special component, "itsType", which is the type of that variable. This provides, among other things, a capability similar to the PL/I like attribute [ANSI76]. Thus

```
var Y : X.itsType
```

declares "Y" to be a variable of the same type as "X".

3.2 Control Structures

All Euclid control structures have explicit end brackets, which eliminates the dangling-else problem of Pascal. The elseif construct serves to counter the plethora of end brackets required to terminate nested conditional statements.

Pascal's while and repeat structures are replaced by a loop structure with conditional or unconditional exit statements that can appear anywhere. The for statement is also enhanced to provide more general iteration control such as iterating through a set.

The goto and label of Pascal were eliminated in Euclid as being undisciplined constructs that presented severe verification problems. The new loop structure, however, together with escape statements, provides a natural way of expressing many of the exit conditions that resulted in the use of gotos in Pascal (see Section 1.3 above).

3.3 Constants

Constant definitions in Euclid are more general than those of Pascal in three ways:

(1) Constants need not be manifest.
(2) Constants can be defined by expressions, whereas Pascal only allows optionally signed values.
(3) Constants of structured types can be defined, whereas Pascal constants are restricted to simple types.

Both the procedural and the declarative aspects of Euclid programs benefit considerably from these extensions.

One dangerous aspect of structured constant definitions is the purely positional notation used to list the components: the

reliance on ordering is both error prone and inconsistent with the named components of records. The constant definition in

```
type ComplexInt
     = record var Re,Im:signedInt end ComplexInt
const Z:ComplexInt := (0,5)
```

would be better expressed as

```
const Z:ComplexInt := (Re:=0, Im:=5)
```

3.4 Variable Bindings

Euclid allows an identifier to be bound to part of an existing variable. For example, the fifth element of the array declared by

```
var A : array 1..100 of T
```

could be renamed as "X" by the variable binding

```
bind var X to A(5)
```

Disallowing the use of the renamed variable within the scope of the binding prevents this feature from introducing aliasing problems. Variable bindings provide a more flexible and readable alternative to the Pascal with statement. For example, due to naming conflicts, a with cannot be used to open two record variables of the same type.

3.5 Operator Precedence

The Pascal operators are (in order of increasing binding strength, with equal priority operators on the same line):

```
=, <>, <, <=, >, >=, in
unary +, binary +, unary -, binary -, or
*, /, div, mod, and
not
```

In his assessment of Pascal Wirth describes the choice of precedence level as "ill-advised" [Wirth75]. Euclid separates out the logical operators and the unary operators, and the precedence levels are:

```
->
or
and
not
=, not =, <, <=, >, >=, in, not in
binary +, binary -, xor
*, div, mod
unary -
```

This more natural classification allows one to write, for example,

```
A<0 or A>N
```

without the parentheses required in Pascal, i.e.

```
(A<0) or (A>N)
```

In addition to rearranging the precedences, Euclid provides conditional logical and, or, and "->" operations that evaluate their right operand only when necessary. This obviates some nesting of conditionals. In Pascal, proper programs must not rely on the implementation evaluation strategy.

References

[ANSI76]
American National Standards Institute; American National Standard Programming Language PL/I, x3.53 (1976).

[Habermann73]
A.N. Habermann; Critical Comments on the Programming Language Pascal; Acta Informatica 3 (1973) pp. 47-57.

[Hoare69]
C.A.R. Hoare; An Axiomatic Basis for Computer Programming; CACM 12,10 (October 1969) pp. 576-580,583.

[Hoare and Wirth73]
C.A.R. Hoare and N. Wirth; An Axiomatic Definition of the Programming Language Pascal; Acta Informatica 2 (1973) pp. 335-355.

[Wirth75]
N. Wirth; An Assessment of the Programming Language Pascal; Proceedings International Conference on Reliable Software, SIGPLAN Notices 10,6 (June 1975) pp. 23-30.

EUCLID AND MODULA

David T. Barnard, W. David Elliott, and David H. Thompson

Abstract: Both Euclid and Modula are programming languages based on Pascal and intended for writing system software such as operating system kernels. The further goals of each language, however, resulted in two rather different languages. Modula is meant to be used in multiprogramming systems primarily on minicomputers; thus Modula aims for very small run-time support and efficient compilation by a small compiler. Many of the Euclid language design decisions, on the other hand, were influenced by the authors' overriding concern for the ability to verify Euclid programs. This paper discusses design goals of the two languages and the language differences that resulted. After contrasting individual features of the two languages, modules and multiprogramming are discussed in more detail.

Contents

1 Overview of Modula

The programming language Modula was designed primarily for programming stand-alone systems for operating existing machines and their peripherals. ([Wirth77a, 77b, and 77c] provide a commendably readable and informative description of the language itself and the rationale behind its design.) Such systems require two additional sets of features not found in general purpose programming languages in the past: features for multiprogramming and features for operating peripheral devices. Such facilities are inherently machine- and configuration-dependent, and as such are not easily described abstractly. Modula attempts to encapsulate such machine dependencies in modules, from whence its name.

The Modula project was started to gain experience in the field of multiprogramming and device handling, with the intent of establishing a discipline for effective and reliable multiprogramming systems design. This effort resulted in a new language, small and very much like Pascal. Its goals of efficient compilation, a simple compiler, and very small run-time support seem to have been met.

The sequential operations of Modula rely very strongly on Pascal. The two notable areas of change from Pascal are additions: the module structure (an encapsulation mechanism similar to the Euclid module) and multiprogramming features (processes, interface modules, and signals). After contrasting individual features of the two languages, modules and multiprogramming are discussed in more detail.

2 General Comparison of the Two Languages

2.1 Data Types

Both Modula and Euclid offer a small set of standard types. Both provide signed integers, Booleans, enumerations, and ordered sets of implementation-defined characters. In addition to signed integers, Euclid also provides unsigned integers and subranges of integers, the bounds of which are implementation defined. Both have deleted reals from Pascal's set of standard types. Euclid provides two additional standard simple types, "StorageUnit" and "AddressType", intended to be used in storage allocation.

Euclid provides sets over any simple type (enumerated and subrange types, as well as the standard simple types). Modula provides something much simpler to implement: "bits"; the number of bits in a variable of type "bits" is intended to be one less than the word size of the target machine. Since the base type of Euclid's sets can be an arbitrary simple type, sets may have an arbitrary positive number of elements (up to some implementation limit), and the base type of the set may have a non-zero lower bound. Thus in general the process of code generation for

operations on "bits" is simpler than code generation for sets in Euclid.

Both Modula and Euclid offer two structuring methods for their standard data types: arrays and records. (Modules provide somewhat more than simple data structuring and are discussed in Section 3.) Euclid offers a packed option for arrays and records, which Modula does not.

Arrays may be explicitly multidimensional in Modula. Euclid provides only one-dimensional arrays, but these arrays may have arrays as their base type. Thus, the Modula declaration

array 0:3, 2:6 of Boolean

corresponds to the Euclid declaration

array 0..3 of array 2..6 of Boolean

Records in the two languages differ in that Modula prohibits the Pascal variant record, while Euclid extends it. Even the nonvariant portions of records are not identical, however. Euclid allows constants as fields of records, whereas Modula does not.

Modula has no mechanism for parameterizing types, while Euclid does. The cost of providing parameterized types seems quite high.

Euclid and Modula have somewhat different notions of type compatibility. Although [Wirth77a] does not spell out what type equality in Modula means, Modula has implemented a very straightforward rule: any two distinctly named types are unequal. Moreover, any anonymous record is unequal to any other record type. In order for a value to be assigned to a variable or a variable to be bound to an identifier, the types must be the same. In Euclid a type identifier is considered an abbreviation for its definition. Except for modules, after all such abbreviations have been removed, two types are the same if their definitions look the same. Any module type or type exported from a module is considered to be different from any other type. Except for a minor variation to accommodate parameterized types, in order for a value to be assigned to a variable or a variable to be bound to an identifier, the types must be the same.

Euclid provides built-in operations on types as well as values of a type, while Modula provides built-in operations only on values. Both languages have a similar set of such operations, e.g., "inc" ("Succ" in Euclid), "dec" ("Pred"), "low" ("first"), "high" ("last"), "integer" ("Ord"), "char" ("Chr"), and "size".

2.2 Variable Declarations and Scopes

Variable declarations in the two languages are quite similar. Declarations must appear before use, although Euclid also provides a *forward* option for mutually referencing declarations.

Scoping methods in the two languages differ quite a bit. Modula conforms to the well-known Algol 60 formula: scopes coincide with blocks. Euclid's open and closed scopes provide greater flexibility, with the consequent added expense to the compiler. Open scopes start following the keyword beginning a structured statement, and declarations may appear almost anywhere in a Euclid program. Statement sequences requiring declarations in Modula must be declared in a procedure or module before use.

Variables of any type may be initialized in either language. In Euclid any initialization is part of the declaration statement, while in Modula an initialization part optionally fcllows the declaration section of each block.

Both languages further constrain Algol 60 inheritance rules for closed scopes with import and export lists (called *use* and *define* lists in Modula). For modules, both languages have similar rules for importing and exporting identifiers - only names on the corresponding list can go in or out. (Euclid excepts *pervasive* identifiers, and Modula excepts standard identifiers.) Modula goes further by disallowing access to any of the structural information about an exported identifier. For routines, which can only import identifiers, the two languages differ only in the case where the optional imports (use) list is omitted. Otherwise, anything not pervasive, a parameter, or explicitly imported is inaccessible. In Euclid, omitting the import list causes no identifiers to be imported; in Modula, normal Algol 60 inheritance rules apply (unlike for modules).

2.3 Control Constructs

For the most part, Euclid and Modula provide an identical set of control constructs. Both have deleted *goto* from their Pascal base, and both have slightly modified most of the other constructs. Both have terminating *end* brackets on control constructs. The if-then-else construct is the same in both languages. Both have added an *elseif* clause (*elsif* in Modula).

The *case* statements in both languages are improvements over Pascal's version. Alternatives are explicitly delineated by terminating keywords, thus solving the parsing problem and improving readability. Modula uses *begin* and *end* to delimit case alternatives, while Euclid uses "=>" and *end*. Euclid requires that either each value of the case type be explicitly mentioned in exactly one alternative or an *otherwise* clause be used in the case statement. Euclid also provides a discriminating case that must be used in evaluating variant record parts. This feature is not needed in Modula, since variant records have been eliminated.

The iteration construct shows a minor but interesting difference in the two languages. Modula provides three statements for iteration, of which the first two are syntactic sugar for the third: while, repeat until, and loop with multiple (single-level) exits. Euclid has a loop statement that is equivalent to the loop of Modula, but somewhat more wordy. The Modula statement

```
loop
when B do Found := false exit
StatementSequence
end
```

requires an if rather than an exit when in Euclid:

```
loop
if B then
     Found := False
     exit
end if
StatementSequence
end loop
```

Although admittedly the while and repeat until statements are trivial to implement in terms of loop constructs, the addition of the two extra constructs seems an unnecessary concession to the Pascal tradition, especially given Modula's goal of a small, simple compiler.

2.4 with Statements and Aliasing

Modula follows Pascal in its use of the with statement. The with statement is used to select a variable of a record type, so that within a statement sequence, the qualifiers of the variable need not be given, but rather just the field names. This is more than just syntactic shorthand for the typing ease of the programmer, since the compiler need only perform the necessary address calculations for this variable once.

In their concern about aliasing, however, the designers of Euclid went somewhat further. Rather than provide simply a mechanism where one can "open" a specific variable of a particular record type and then refer to the field names without further qualification, Euclid provides a mechanism for renaming a variable at the beginning of a scope. This new variable name then exists for the duration of the scope. Address calculations can still be performed only once, at the beginning of the scope. Perhaps more importantly, more than one variable of the same record type can be effectively "open" simultaneously. Since the new names must be distinct from other names known in the enclosing scope, no naming conflicts can occur. Moreover, this renaming is not restricted to record variables.

2.5 Procedures and Functions

The bodies of procedures and functions are similar in the two languages except for the differences already mentioned, such as name inheritance and scopes in Euclid not necessarily coinciding with blocks. Other major differences in routines are side effects in functions and parameter association

Euclid functions may not have side effects. Modula does not prevent functions from exerting side effects, although functional side effects are discouraged as poor programming practice, and no explicit use is made of this in the defining Modula documents. Moreover, if the use list is omitted in Modula, the amount of information that must be maintained to ensure the absence of functional side effects can be excessive.

It is illegal in Euclid to both import an identifier and access it via another name through the parameter list, as in:

```
var C: signedInt
...
procedure P (var A: signedInt) =
     imports (C)
     begin
     ...
     end
...
P(C)
```

Modula does not prohibit this type of aliasing.

2.6 Constants

In Modula all constants are scalar, with one exception: one may represent a constant of type "bits". In Euclid, on the other hand, constants may be declared of any type.

All constants in Modula are manifest: their values must be determinable at compile time. As a result, storage requirements for each module and procedure are known at compile time. In Euclid a constant need not be compile-time evaluable; rather, it is bound each time its scope is entered. The difference is thus a matter of binding time.

2.7 Syntactic Issues

In order to denote a statement sequence in Pascal, a begin block is required. Both Modula and Euclid provide simpler statement structures in general than Pascal. Both take the view that where one statement can go, many can. Both languages have a recursive definition of statement requiring no additional keywords, which is made possible by the terminating keyword on all control constructs. Thus,

```
while B do begin S1; S2 end
```

in Pascal becomes

```
while B do S1; S2 end
```

or

```
loop when not B exit S1; S2 end
```

in Modula, and

```
loop; exit when not B; S1; S2 end loop
```

in Euclid.

Another interesting improvement that Modula and Euclid share over most other languages, although not Pascal, is the use of semicolons as statement delimiters. Modula uses the semicolon as a separator rather than a terminator, in both record field lists and statements. However, since both record fields and statements may be empty, according to the Modula grammar, it does not matter whether one actually writes the semicolon after the final field or statement. Euclid goes even further in this respect: Euclid has a convention whereby the scanner for the language inserts semicolons for the parser's and the error recovery mechanism's benefit at the end of lines, depending on surrounding keywords. Thus the user need type very few semicolons. The principal problem with such a convention is that the user will very quickly get into the habit of not placing a semicolon anywhere. When a semicolon omission error does occur, the explanation that only some statements need semicolons, while others do not, will seem quite inconsistent.

3 Modules

Both the Modula and Euclid modules are descendants of the Simula 67 class [Dahl et al.68] that add information hiding capabilities to the encapsulation and data retention provided by the Simula 67 class. In contrast to the Euclid module, which is principally intended as a data abstraction mechanism (see the "Abstract Data Types in Euclid" paper), the Modula module is principally intended as a fence that establishes a static scope of identifiers whose visibility can be controlled. The distinction between the two notions of modules will be drawn more clearly after a discussion of data retention, visibility of identifiers, and storage allocation in Modula.

3.1 Data Retention in Modula

The Modula module allows local enclosed procedures to share retained protected data objects. A module comes into existence when the enclosing procedure (or process) is called, and vanishes when that procedure invocation is completed. Objects declared within a module are considered local to the enclosing procedure.

A module's data objects are thus retained after termination of any procedures enclosed within that module.

3.2 Visibility of Identifiers in Modula

Like Euclid, Modula provides for explicit control of the visibility of identifiers. The Modula module has use and define lists, directly analogous to Euclid's import and export lists. In both languages, if a module has no export (define) list, no identifiers are exported; if a module has no import (use) list, nothing is imported (except prevasive identifiers in Euclid and standard identifiers in Modula).

Modula more severely limits access to variables and structural information of the enclosed types than does Euclid. Although it is questionable whether the ability in Euclid to assign to an exported variable is of significant value, Modula goes one step further and causes all exported variables to be read-only outside of the module. The intent is that variables belonging to a module should not actually be exported at all; making them exportable as read-only variables obviates declaring functions that simply yield a variable's value. The capability to export a variable can be simulated, however, by exporting an "Assign" routine that assigns values to that variable.

A module can export only the name of a type in Modula. In particular, no field names are exported for records, and no index ranges or component types are exported for arrays. Thus, the environment external to a module has no way of knowing what the structure of an exported type is.

3.3 Storage Allocation in Modula

There are no unnamed begin blocks in Modula. The keywords begin and end are only used to delimit statement sequences that constitute the actions of procedures and processes, the initialization for a module, or the alternative actions in a case statement. When a local scope is required, the program and data parts must be previously written as a procedure definition and invoked at the desired point. Although no storage allocation need be done except at procedure or process invocation, this prevents any dynamic allocation of storage needed only temporarily.

3.4 Euclid Modules versus Modula Modules

The Modula module is intended to encapsulate such entities as a scanner in a compiler or a disk store manager, i.e., program segments of relatively large size where only a single instance exists. Multiple instances of a Euclid module, on the other hand, can be instantiated as needed. Herein lies the major difference between Euclid and Modula modules.

A Euclid module can, however, be represented as a Modula module declaration. Consider the following Euclid module definition (this example is "borrowed" from [Wirth77c] and translated into Euclid):

```
module M
     exports (P,Q)
     pervasive type T = ...
     var X,Y: T
     procedure P (var U: signedInt) imports (X,Y)
          begin ... end P
     procedure Q (Var V; signedInt) imports (X,Y)
          begin ... end Q
     ...
     initially ...  { initialization of instance }
     end module
```

This is expressed in terms of a Modula module as follows:

```
module M;
     define R,P,Q,S;
     type T = ...
     type R = record X,Y: T end;
     procedure P (var W: R; var U: integer);
          begin ... end P
     procedure Q (var W: R; var V: integer);
          begin ... end Q;
     ...
     procedure S (var W: R);
          begin ...  (* initialization of instance *)
          end S;     (* must be explicitly invoked
                        for each instance *)
     end M;
```

Note that the variables local to the Euclid module become record components of a new record type in the Modula module.

The advantages of the Euclid module include (1) some syntactic convenience in specifying procedure invocation, (2) ease in naming procedures in that the same name may be used inside different Euclid modules, and (3) initialization not needing to be invoked explicitly (as in procedure "S" in the Modula example). These are not, however, major differences, especially in the Modula application area, where Wirth claims it seems natural to have unique names for all operators exported from modules [Wirth77c, p.69].

Although Modula modules were not intended as a data abstraction mechanism, the problems in using them as such are not inherent in the nature of modules, but rather due to other language decisions. In particular, the lack of parameterized types and the requirement that all array bounds be compile-time evaluable make Euclid's parameterized module instantiations at best unwieldy to mirror in Modula. Consider the possible need for a parameterized queue module where the length is to be specified as a parameter supplied at instantiation. This is

provided in Euclid in a straightforward manner, whereas Modula could only provide for a limited number of cases, since each particular maximum length would require a separate complete module definition, with unique names for all operators.

4 Multiprogramming

Modula is explicitly designed to attack the domination of assembly language programming in systems for operating particular machines. Such systems require facilities for multiprogramming and for operating peripheral devices. There are no such features explicitly provided in Euclid, although both languages use modules to encapsulate machine dependencies. Modula provides processes, send and wait on signals as synchronization primitives, a form of monitors for mutual exclusion, and device processes. This section explains these features of Modula and indicates how they could be provided in Euclid.

4.1 Processes

A process in Modula is a syntactic unit that must be defined and invoked at the outermost (main program) level. This means that a process cannot spawn son processes, although there can be multiple instances of a process. A process is given a fixed size memory, so if a recursive procedure is used in a process, a compiler directive must indicate the maximum depth of recursion. Accordingly, when the end of the main program is reached, all processes have been started, and an implicit guarantee can be made that storage overflow will not occur. The storage belonging to a process that has terminated need not be reused since no further storage can be allocated. There is thus no need for a dynamic storage allocation scheme or for the management of sons that might outlive their fathers.

A process can be in one of three states: ready, running, or waiting on a signal. All processes that are ready or running are linked into a ring. Scheduling will be discussed below.

4.2 Mutual Exclusion

Mutual exclusion is provided in Modula by a special kind of module called an interface module, which is distinguished by the keyword interface. Only one process can be actively executing within an interface module at any point in time, as with Hoare's monitors [Hoare74].

There are two differences with Hoare's monitors. The first is that different interface modules can access common external variables, although such variables must be explicitly imported. The second (somewhat lesser) difference is that the internal data structures of an interface module can be exported, although this is strongly discouraged as a programming practice. Such exported

variables are read-only; any updating must be accomplished via interface module procedures.

4.3 Synchronization

Synchronization of processes is accomplished by signals, which correspond to Hoare's conditions. Signals syntactically appear as variables, but do not have values in the normal sense and are not assignable. They are operated on only by the system functions "wait", "send", and "awaited".

A signal has an associated queue of waiting processes. "wait" causes a process to be inserted into the appropriate queue. A rank can be specified, and waiting processes are ordered according to the rank. "send" allows the first process waiting in the queue (if any) to continue. When a process waits on or signals a condition, it steps out of the domain of mutual exclusion. "awaited" simply tests whether a particular queue is ncnempty.

Mutual exclusion comes for free if a single processor is used without forced time-slicing. Processor switching only occurs at "send" and "wait" statements, and thus there is no implicit processor switching (that is, processor switching that occurs at arbitrary times without an explicit process directive). (Ccmplications due to I/O are discussed later.) "send" can cause a waiting process to gain control, while the sender simply goes into the ring of ready processes. "wait" causes the waiting process to be put on the appropriate queue, and a process from the ring to be scheduled.

4.4 A Synchronization Example

We present an example illustrating the synchronization facilities discussed above. This example shows two processes communicating through a circular buffer. The process "Producer" reads characters from an input device and stores them in the buffer. The process "Consumer" removes characters from the buffer and prints them on an output device. The system is non-terminating. The procedures "Read" and "Print" are assumed to be defined in the enclosing scope.

```
module Listinput;
    use Read, Print;
    interface module Bufferhandling;
        define Get, Put;
        const Nmax=256;
        var N, In, Out: integer;
            Nonempty, Nonfull: signal;
            Buf: array 1:Nmax of char;
```

```
        (* Insert a character into the circular buffer *)
        procedure Put (Ch: char);
            begin
                if N=Nmax then wait(Nonfull) end;
                inc(N);
                Buf[In] := Ch;
                In := (In mod Nmax)+1;
                send(Nonempty)
            end Put;

        (* Remove a character from the circular buffer *)
        procedure Get(var Ch: char);
            begin
                if N=0 then wait(Nonempty) end;
                dec(N);
                Ch := Buf[Out];
                Out := (Out mod Nmax)+1;
                send(Nonfull)
            end Get;
        begin
            N := 0;
            In := 1;
            Out := 1
        end Bufferhandling;

    (* Read input characters and store in buffer *)
    process Producer;
        use Read, Put;
        var Ch: char;
        begin
            loop
                Read(Ch);
                Put(Ch);
            end
        end Producer;

    (* Remove characters from buffer and print *)
    process Consumer;
        use Get, Print;
        var Ch: char;
        begin
            loop
                Get(Ch);
                Print(Ch);
            end
        end Consumer;

    begin
        Producer;
        Consumer
    end Listinput;
```

4.5 PDP-11 Device Modules

An implementation of Modula on a specific machine is intended to provide access to I/O devices. The documented implementation of Modula is for the PDP-11 [Wirth77c]. This section describes the device modules that control input and output on the PDP-11.

Device drivers are written as processes within interface modules prefixed by the keyword device. A driver is initiated within the interface module body. An I/O operation is accomplished by an assignment to a device control register, followed by the statement doio. Device control registers are at fixed locations in memory, and these addresses are specified in the declaration of a register identifier.

doio initiates the actual I/O operation and causes the driver to go to sleep waiting for an interrupt. An interrupt on the PDP-11 is associated with an address in memory called the interrupt vector. This address must be specified when the process is declared. When the interrupt occurs, an implicit processor switch takes place: the running process is suspended and the driver takes over until it executes another doio or a "wait", at which time it is suspended and the previously interrupted process regains control. A device process cannot "send" a signal to another device process. If a device process does "send" a signal, the receiver is marked ready and inserted into the ring, and the device process continues. The device process should not invalidate the signalled condition, but this is not checked.

There are only single instances of device processes. Device processes can be in the states running, waiting for signal, or waiting for interrupt. They do not get into the ring of ready user processes.

4.6 Multiprogramming in Euclid

There are no mechanisms provided for multiprogramming in Euclid; whatever features are desired must be written in the language itself. We will consider the problem of providing mechanisms in Euclid similar to those available in Modula.

Since processes cannot be explicitly designated, a procedure "Spawn" must be written that will instantiate a procedure as a process. When given the machine address of a procedure, "Spawn" must provide it with stack space and link a descriptor for it into the ring. Since neither memory requirements nor a descriptor are provided by the compiler, they would have to be explicitly passed as parameters. "Spawn" must have access to machine registers to set up the storage linkage.

A Euclid module could play the role of a Modula interface module if we assume a single processor and switching only at the request of the running process (except for interrupts), as Modula does. To allow external routines to access the local data, all

identifiers defined in the module would be explicitly exported. Signals could be implemented as pointers, as in Modula, and "Send" and "Wait" procedures could be provided. Again, these would have to be in a machine dependent module as they must have access to processor status registers to provide processor switching.

Providing multiprogramming facilities requires manipulation of run-time environments (providing stack space, linking procedure descriptors, storing process state vectors, etc.). In Euclid the required information is known only to the compiler. In order to program the facilities we have discussed, it is necessary to use machine code in machine-dependent modules to access all the hardware registers, and to have a complete knowledge of how a particular Euclid compiler implements the run-time environment. This means that the language (by itself) is not sufficient for writing operating system kernels.

4.7 Evaluation of Multiprogramming Facilities

The simple multiprogramming facilities of Modula are easy to understand and to implement; the Modula runtime support package is less than 200 bytes of code. However, its simplicity can lead to problems. Consider the situation where two processes are active, one of which is doing a lot of data transfers, the other a lot of computing. To maximize use of machine resources we would prefer interrupts from completed data transfers to be quickly serviced by the I/O process so that it could initiate another transfer and thus keep the device busy. In Modula, the I/O process would be suspended when the first transfer started, the compute process would take over and be interrupted when the transfer was complete, and then the compute process would be allowed to continue after the driver had handled the interrupt. Because there is no processor pre-emption, the I/O process could wait for a long time before being rescheduled to start the next I/O operation. This defect is not critical in an experimental system like Modula because processes could periodically issue explicit "wait"s to allow such processor switching. However, this relies on the honorable intentions of all processes.

Euclid does not have any multiprogramming facilities, although those of Modula could be provided as discussed above. Additional parameters would be required for processes, and the programmer of the machine dependent module would have to have knowledge of the compiler's model of the entire runtime environment. The major advantage of Modula in this area is the provision of an extremely efficient and simple set of multiprogramming facilities as part of the language itself. The major advantage of Euclid is the flexibility of allowing different multiprogramming facilities to be provided, whereas Modula is locked into a particular multiprogramming methodology. The advantage here only arises because of access to machine code from Euclid - no useful facilities or primitives are provided in the language.

5 Summary

Although both Euclid and Modula are based on Pascal and intended for writing system software, the further goals of the languages resulted in two rather different languages. In general, Modula is a much smaller and simpler language than Euclid. Modula has perhaps more specific goals: it is intended for multiprogramming, especially on minicomputers. Euclid is a much more complex language, due in part to its more encompassing goals and in particular to its goal of verifiability. More mechanism is included in the language so that more errors can be caught by the compiler and so that more of the information needed for verification is explicit in the program text. Quite often, where Modula deleted or restricted a feature in its Pascal base, Euclid extended it. Thus the task of designing and implementing a reliable compiler for Modula seems quite simple compared to the effort required for a reliable Euclid compiler.

It seems almost simplistic to state that we would choose to use each language in applications for which the respective language was intended. This statement, however, has some subtle consequences. Where multiprogramming on minis or verification is not the overwhelming concern, the question then becomes which language would provide greater confidence that a program written in it would indeed do the job at hand.

One requires more than faith that an implementation is correct according to its specification. Modula provides a great amount of faith precisely because of the size and simplicity of the language and its compiler. Assuming there existed a reliable Euclid compiler, Euclid, neither small nor simple, provides specific features such as type safety and aliasing restrictions that serve as a firm basis for rigorously verifying the correctness of a program. Moreover, even for programs not intended to be formally verified, the philosophy embodied in Euclid increases the programmer's confidence that the program is intuitively correct, and thus meets its specification.

References

[Dahl et al.68]
O.-J. Dahl, B. Myhrhaug, and K. Nygaard; The Simula 67 Common Base Language; Norwegian Computing Centre, Oslo (1968).

[Hoare74]
C.A.R. Hoare; Monitors: an Operating System Structuring Concept; CACM 17,10 (October 1974) pp. 549-557.

[Wirth77a]
N. Wirth; Modula: a Language for Modular Multiprogramming; Software, Practice and Experience 7,1 (1977) pp. 3-35.

[Wirth77b]
N. Wirth; The Use of Modula; Software, Practice and Experience 7,1 (1977) pp. 37-65.

[Wirth77c]
N. Wirth; Design and Implementation of Modula; Software, Practice and Experience 7,1 (1977) pp. 67-84.

INDEX TO THE EUCLID REPORT

UNIVERSITY OF TORONTO

COMPUTER SYSTEMS RESEARCH GROUP

BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
J.J. Horning and W.R. Lalonde, September 1970
[ACM SIGPLAN Notices, November 1970]

CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
W.R. Lalonde, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
J.D. Gorrie, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-4 DYLAN USER'S MANUAL
P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
Alan C.M. Brown and J.J. Horning, March 1971

CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
Richard C. Holt, April 1971
[Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
John Neill Thomas Potvin, August 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED ANIMATION SYSTEM
Kenneth B. Evans, January 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
William Gregg Alexander, February 1972
[M.Sc. Thesis, DCS 1971; Computer, v.8, n.11, November 1975]

CSRG-11 PROJECT SUE STATUS REPORT
J.W. Atwood (ed.), April 1972

* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL
Rupert Bramall, April 1972 [M.Sc. Thesis, DCS, 1971]

* CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD
Lewis R. James, May 1972 [M.Sc. Thesis, DCS, 1972]

+ Abbreviations:
DCS - Department of Computer Science, University of Toronto
EE - Department of Electrical Engineering, University of Toronto
* - Out of print

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING
Kenneth C. Sevcik, May 1972
[Ph.D. Thesis, Committee on Information Sciences, University of Chicago, 1971; JACM, January 1974]

CSRG-15 PROCESS STRUCTURING
J.J. Horning and B. Randell, June 1972
[ACM Computing Surveys, March 1973]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMTION OVERHEAD IS NOT NEGLIGIBLE
Kenneth C. Sevcik, June 1972
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

* CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE
K.C. Sevcik et al, September 1972
[Proceedings AFIPS Fall Joint Computer Conference, v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN
David B. Wortman, December 1972
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING
R. Kvaternik, December 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS
G.G. Kalmar, January 1973 [M.Sc. Thesis, DCS, 1972]

CSRG-23 COMPILER STRUCTURE
W.M. McKeeman, January 1973
[Proceedings of the USA-Japan Computer Conference, 1972]

* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
Eleanor A. Lester, April 1973 [M.Sc. Thesis, DCS, 1973]

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS: AN INITIAL EXPERIMENT
Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON THE REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
Marc Louis Joliat, October 1973 [Ph.D. Thesis, EE 1973]

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
Henry John Pasko, December 1973 [M.Sc. Thesis, DCS 1973]

* CSRG-31 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), Second Edition, March 1974

CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
E.D. Lazowska, May 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
F. Lochovsky and D. Tsichritzis, May 1974 [INFOR, to appear]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
P. Bernstein and D. Tsichritzis, May 1974 [Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974 [Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS
Laurence M. Weissman, August 1974
[Ph.D. Thesis, DCS, 1974]

* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
David M. Lasker, September 1974 [M.Sc. Thesis, DCS, 1974]

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
Glenn F. Stewart, September 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
J. Klebanoff, F. Lochovsky, A. Rozitis, and D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF RELIABLE SOFTWARE
David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
B.L. Clark and F.J.B. Ham, September 1974

CSRG-43 A DATA BASE PROCESSOR
E.A. Ozkarahan, S.A. Schuster and K.C. Smith, November 1974 [Proceedings National Computer Conference 1975, v.44, pp.379-388]

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A COMPUTING ENVIRONMENT
Eric C.R. Hehner, November 1974 [Ph.D. Thesis, DCS, 1974]

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN, DYADIC SPECIFICATION, COMPLEMENTARY SEMANTICS
J.E. Donahue, J.D. Gannon, J.V. Guttag and J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM DECISION TABLES
Helmut Schumacher, December 1974
[M.Sc. Thesis, DCS, 1974]

CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
John D. Gannon, January 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
Christopher J.M. Turnbull, January 1975
[Ph.D. Thesis, EE, 1974]

* CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
D. Tsichritzis, February 1975 [in Data Base Description, Dongue and Nijssen (eds.), North Holland Publishing Co.]

* CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
M. Brodie (ed). February 1975 [Proceedings Pacific ACM Conference, 1975]

CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS
David T. Barnard, March 1975 [M.Sc. Thesis, DCS, 1975]

* CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.V. Guttag (ed.), Third Edition, April 1975

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
Richard C. Holt and David B. Wortman, May 1975

CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

* CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL
Hans Albrecht Schmid and J. Richard Swenson, July 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]

* CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975
[Ph.D. Thesis, DCS, 1975]

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEURISTICS
Lazlo Sugar, December 1975 [M.Sc. Thesis, DCS, 1975]

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith, February 1976 [Proceedings National Computer Conference 1976, v.45, pp.855-862]

* CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, February 1976 [ACM Transactions on Database Systems, v.1, n:4, December 1976]

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976
[M.Sc. Thesis, DCS, 1975]

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1976 [M.Sc. Thesis, DCS, 1976]

* CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L.Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco
April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard and D. Thompson (Eds.), Fourth Edition, May 1976

* CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D. Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A DATA BASE MACHINE
E.A. Ozkarahan and K.C. Sevcik, May 1976

* CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow, R. Baker and S. Pozgaj, July 1976

CSRG-73 AN ALGORITHMIC APPROACH TO NORMALIZATION OF RELATIONAL DATA BASE SCHEMAS
P.A. Bernstein and C. Beeri, September 1976

CSRG-74 A HIGH-LEVEL MACHINE-ORIENTED ASSEMBLER LANGUAGE FOR A DATA BASE MACHINE
E.A. Ozkarahan and S.A. Schuster, October 1976

CSRG-75 DO CONSIDERED OD: A CONTRIBUTION TO THE PROGRAMMING CALCULUS
Eric C.R. Hehner, November 1976

CSRG-76 "SOFTWARE HUT": A COMPUTER PROGRAM ENGINEERING PROJECT IN THE FORM OF A GAME
J.J. Horning and D.B. Wortman, November 1976

CSRG-77 A SHORT STUDY OF PROGRAM AND MEMORY POLICY BEHAVIOUR
G. Scott Graham, January 1977

CSRG-78 A PANACHE OF DBMS IDEAS
D. Tsichritzis, February 1977

CSRG-79 THE DESIGN AND IMPLEMENTATION OF AN ADVANCED LALR PARSE TABLE CONSTRUCTOR
David H. Thompson, April 1977 [M.Sc. Thesis, DCS, 1976]

CSRG-80 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard (Ed.), Fifth Edition, May 1977

CSRG-81 PROGRAMMING METHODOLOGY: AN ANNOTATED BIBLIOGRAPHY FOR IFIP WORKING GROUP 2.3
Sol J. Greenspan and J.J. Horning (Eds.), First Edition, May 1977

CSRG-82 NOTES ON EUCLID
edited by W. David Elliot and David T. Barnard, August 1977